REPORT DOCUMENTATION PAGE

Form Approved
OPM No 0704-0188

AD-A265 014

...average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data ...garding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington ...rts, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

2. REPORT DATE

3. REPORT TYPE AND DATES COVERED
Final: 19 Mar 93

4. TITLE AND SUBTITLE
Validation Summary Report: Digital Equipment Corporation, DEC Ada for OpenVMS VAX Systems, Version 3.0-7, VAXstation 4000 Model 60 (host & target), 930319S1.11316

5. FUNDING NUMBERS

6. AUTHOR(S)
National Institute of Standards and Technology
Gaithersburg, MD
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
National Institute of Standards and Technology
National Computer Systems Laboratory
Bldg. 255, Rm A266
Gaithersburg, MD 20899 USA

8. PERFORMING ORGANIZATION REPORT NUMBER
NIST93DEC505_2_1.11

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, RM 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
Digital Equipment Corporation, DEC Ada for OpenVMS VAX Systems, Version 3.0-7, VAXstation 4000 Model 60 (under VMS Version 5.5) (host & target), ACVC 1.11

93-11590

14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

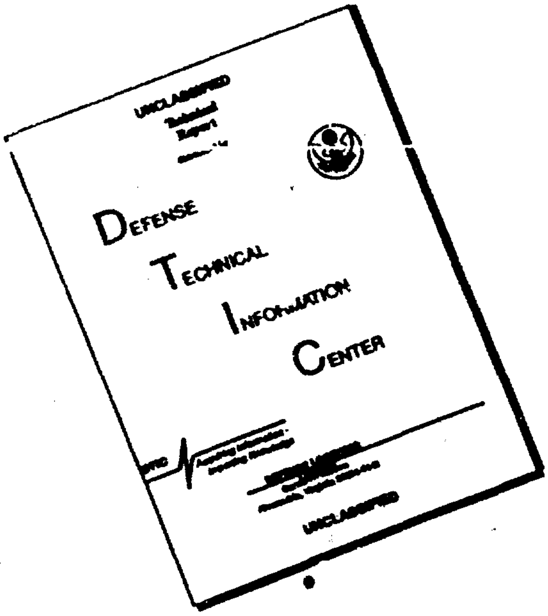
16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

NSN 7540-01-280 550

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-128

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

AVF Control Number: NIST93DEC505_2_1.11
DATE COMPLETED
BEFORE ON-SITE: 93-02-09
AFTER ON-SITE: 93-03-22
REVISIONS:

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 930319S1.11316
Digital Equipment Corporation
DEC Ada for OpenVMS VAX Systems, Version 3.0-7
VAXstation 4000 Model 60 =>
VAXstation 4000 Model 60

Prepared By:
Software Standards Validation Group
Computer Systems Laboratory
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899

AVF Control Number: NIST93DEC505_2_1.11

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on March 19, 1993.

Compiler Name and Version:	DEC Ada for OpenVMS VAX Systems, Version 3.0-7
Host Computer System:	VAXstation 4000 Model 60 under VMS Version 5.5
Target Computer System:	VAXstation 4000 Model 60 under VMS Version 5.5

See section 3.1 for any additional information about the testing environment.

→ 930319S1.11316

As a result of this validation effort, Validation Certificate ~~<CERTIFICATE NUMBER>~~ is awarded to Digital Equipment Corporation. This certificate expires 2 years after ANSI/MIL-STD-1815B is approved by ANSI.

This report has been reviewed and is approved.

Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division (ISED)

Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
Validation Group

Computer Systems Laboratory (CSL)
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg MD 20899

for

Ada Validation Organization
Director, Computer & Software
Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Customer: Digital Equipment Corporation

Certificate Awardee: Digital Equipment Corporation

Ada Validation Facility: National Institute of Standards and Technology
Computer Systems Laboratory (CSL)
Software Validation Group
Building 225, Room A266
Gaithersburg, Maryland 20899

ACVC Version: 1.11

Ada Implementation:

Compiler Name and Version: DEC Ada for OpenVMS VAX Systems, Version 3.0-7

Host Computer System: VAXstation 4000 Model 60 under VMS Version 5.5

Target Computer System: VAXstation 4000 Model 60 under VMS Version 5.5

Declaration:

I the undersigned, declare that I have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.

Charles Z Mitchell 3/25/93

Customer Signature Date
Company Digital Equipment Corporation
Title: Project Manager

Certificate Awardee Signature Date

e) Non-Processed Floating-Point Precision Tests — 0

f) Total Number of Inapplicable Tests — 79 (c+d+e)
g) Total Number of Tests for ACVC 1.11 — 4170 (a+b+f)

3.3 TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded onto VAX 6000 Model 350 and transferred to the host/target computer by Ethernet.

After the test files were loaded onto the host/target computer, the full set of tests was processed by the Ada implementation.

The tests were compiled, linked, and executed on the host/target computer system, as appropriate. The results were captured on the host/target computer system. The test results were transferred from the host/target computer to the VAX 6000 Model 350 where a magnetic was written capturing the test results.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

```
ADA/COPY_SOURCE/NODEBUG/NODIAG/ERROR_LIMIT=1000/LIST/NOSHOW
```

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 3.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, Validation consisting of the test suite, the support programs, the ACVC Capability user's guide and the template for the validation summary (ACVC) report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification Office system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC an Ada version.

Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user- written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable Test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.

Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn Test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

Some tests are withdrawn by the AVO from the ACVC because they do not conform to the Ada Standard. The following 95 tests had been withdrawn by the Ada Validation Organization (AVO) at the time of validation testing. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 91-08-02.

E28005C	B28006C	C32203A	C34006D	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	B41308B	C43004A
C45114A	C45346A	C45612A	C45612B	C45612C	C45651A
C46022A	B49008A	B49008B	A74006A	C74308A	B83022B
B83022H	B83025B	B83025D	B83026B	C83026A	C83041A
B85001L	C86001F	C94021A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
AD7206A	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. The inapplicability criteria for some tests are explained in documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

C24113W..Y (3 tests) use a line length in the input file which exceeds 255 characters.

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT; for this implementation, there is no such type.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is TRUE.

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

B91001H checks that an address clause may not precede an entry declaration; this implementation does not support address clauses for entries. (See section 2.3.)

C96005B uses values of type DURATION's base type that are outside the range of type DURATION; for this implementation, the ranges are the same.

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than was specified by the length clause, as allowed by AI-00558.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions; this implementation provides no package MACHINE_CODE.

The 18 tests listed in the following table check that USE_ERROR is raised if the given file operations are not supported for the given combination of mode and access method; this implementation supports these operations.

Test	File Operation	Mode	File Access Method
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO

CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

The tests listed in the following table check the given file operations for the given combination of mode and access method; this implementation does not support these operations.

Test	File Operation	Mode	File Access Method
CE2105A	CREATE	IN_FILE	SEQUENTIAL_IO
CE2105B	CREATE	IN_FILE	DIRECT_IO
CE3109A	CREATE	IN_FILE	TEXT_IO

The following 12 tests check operations on sequential, direct, and text files when multiple internal files are associated with the same external file and one or more are open for writing; USE_ERROR is raised when this association is attempted.

CE2107B	CE2107E	CE2107G	CE2110B	CE2110D
CE2111D	CE2111H	CE3111B	CE3111D..E	CE3114B
CE3115A				

CE2107C..D (2 tests), CE2107H, and CE2107L apply function NAME to temporary sequential, direct, and text files in an attempt to associate multiple internal files with the same external file; USE_ERROR is raised because temporary files have no name.

CE2108B, CE2108D, and CE3112B use the names of temporary sequential, direct, and text files that were created in other tests in order to check that the temporary files are not accessible after the completion of those tests; for this implementation, temporary files have no name.

CE2111C checks that a supplied mode parameter can be RESET from IN_FILE to OUT_FILE (An amplification in accessing privileges while the external file is being accessed). The proper exception is raised.

CE2203A checks that WRITE raises USE_ERROR if the capacity of an external sequential file is exceeded; this implementation cannot restrict file capacity.

CE2401H, EE2401D, and EE2401G use instantiations of DIRECT_IO with unconstrained array and record types; this implementation raises USE_ERROR on the attempt to create a file of such types.

CE2403A checks that WRITE raises USE_ERROR if the capacity of an external direct file is exceeded; this implementation cannot restrict file capacity.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST; for this implementation, the value of COUNT'LAST is greater than 150000, making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 1 test.

B91001H was graded inapplicable by Evaluation Modification as directed by the AVO. This test expects an error to be cited for an entry declaration that follows an address clause for a preceding entry; but this implementation does not support address clauses for entries (rather, it provides a package that allows a task to wait for the delivery of one or more signals), and so rejects the address clause.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For technical and sales information about this Ada implementation, contact:

Attn: Maryanne Cacciola
Ada Product Manager
Digital Equipment Corporation
110 Spit Brook Road (ZKO2-1/M11)
Nashua, NH 03062
(603) 881-1028

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system -- if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a) Total Number of Applicable Tests	3996
b) Total Number of Withdrawn Tests	95
c) Processed Inapplicable Tests	79
d) Non-Processed I/O Tests	0

e) Non-Processed Floating-Point Precision Tests 0

f) Total Number of Inapplicable Tests 79 (c+d+e)
g) Total Number of Tests for ACVC 1.11 4170 (a+b+f)

When this implementation was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded onto VAX 6000 Model 350 and tranfered to the host/target computer by Ethernet.

After the test files were loaded onto the host/target computer, the full set of tests was processed by the Ada implementation.

The tests were compiled, linked, and executed on the host/target computer system, as appropriate. The results were captured on the host/target computer system. The test results were transferred from the host/target computer to the VAX 6000 Model 350 where a magnetic was written capturing the test results.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

```
ADA/COPY_SOURCE/NODEBUG/NODIAG/ERROR_LIMIT=1000/LIST/NOSHOW
```

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	255-- Value of V
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table contains the values for the remaining macro parameters.

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2_147_483_647
$DEFAULT_MEM_SIZE	2**31-1
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	VAX_VMS
$DELTA_DOC	2.0**(-31)
$ENTRY_ADDRESS	FCNDECL.ENTRY_ADDRESS
$ENTRY_ADDRESS1	FCNDECL.ENTRY_ADDRESS1
$ENTRY_ADDRESS2	FCNDECL.ENTRY_ADDRESS2
$FIELD_LAST	2_147_483_647
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	LONG_LONG_FLOAT
$FORM_STRING	"! Satisfies condition 1 only"
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	75_000.0
$GREATER_THAN_DURATION_BASE_LAST	131_073.0
$GREATER_THAN_FLOAT_BASE_LAST	1.80141E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	1.7014117E+38
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.0E308
$HIGH_PRIORITY	15

$ILLEGAL_EXTERNAL_FILE_NAME1	BAD/CHAR^@.~!
$ILLEGAL_EXTERNAL_FILE_NAME2	x"&(1..256=>'c')&"y
$INAPPROPRIATE_LINE_LENGTH	65_536
$INAPPROPRIATE_PAGE_LENGTH	-1
$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006E1.TST")
$INTEGER_FIRST	-2147483648
$INTEGER_LAST	2147483647
$INTEGER_LAST_PLUS_1	2_147_483_648
$INTERFACE_LANGUAGE	C
$LESS_THAN_DURATION	-75_000.0
$LESS_THAN_DURATION_BASE_FIRST	-131_073.0
$LINE_TERMINATOR	' '
$LOW_PRIORITY	0
$MACHINE_CODE_STATEMENT	NULL;
$MACHINE_CODE_TYPE	NO_SUCH_TYPE
$MANTISSA_DOC	31
$MAX_DIGITS	33
$MAX_INT	2147483647
$MAX_INT_PLUS_1	2_147_483_648
$MIN_INT	-2147483648
$NAME	SHORT_SHORT_INTEGER
$NAME_LIST	VAX_VMS,VAXELN,OPENVMS_AXP,RISC_ULTRIX,MIL_STD_1750A,MC68000,MC68020,MC68040,CPU32
$NAME_SPECIFICATION1	ACVC_LFN_DEVICE:[ACVC_LFN_AREA]X2120A.;1

$NAME_SPECIFICATION2	ACVC_LFN_DEVICE:[ACVC_LFN_AREA]X2120B.;1
$NAME_SPECIFICATION3	ACVC_LFN_DEVICE:[ACVC_LFN_AREA]X3119A.;1
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	1_048_576
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	VAXELN
$PAGE_TERMINATOR	' '
$RECORD_DEFINITION	RECORD NULL; END RECORD;
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	0
$TICK	10.0**(-2)
$VARIABLE_ADDRESS	FCNDECL.VARIABLE_ADDRESS
$VARIABLE_ADDRESS1	FCNDECL.VARIABLE_ADDRESS1
$VARIABLE_ADDRESS2	FCNDECL.VARIABLE_ADDRESS2
$YOUR_PRAGMA	EXPORT_OBJECT

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

Compiler options for DEC Ada hosted on OpenVMS VAX and OpenVMS AXP systems

The default compiler options were used except as follows:

1. /LIST was used to produce compiler listings.

2. /NODEBUG was used to inhibit the generation of debugging information in the object file since such information is not relevant to validation.

3. /ERROR_LIMIT=1000 was used since more than 30 errors are diagnosed for some validation tests.

4. /NOSHOW was used to exclude portability information from the compiler listings

The DEC Ada compiler options and defaults for OpenVMS VAX and OpenVMS AXP systems are summarized as follows:

- /ANALYSIS_DATA or /NOANALYSIS_DATA

  Controls whether a data analysis file containing source code cross-referencing and static analysis information is created. The default is /NOANALYSIS_DATA.

- /CHECK or /NOCHECK

  Controls whether run-time error checking is suppressed. (Use of /NOCHECK is equivalent to giving all possible suppress pragmas in the source program.) The default is /CHECK (error checking is not suppressed except by pragma).

- /COPY_SOURCE or /NOCOPY_SOURCE

  Controls whether the source being compiled is copied into the compilation library for a successful compilation. The default is /COPY_SOURCE.

- /DEBUG or /NODEBUG or /DEBUG=option

  where option is one of

  ALL, SYMBOLS or NOSYMBOLS, TRACEBACK or NOTRACEBACK, or NONE

  Controls the inclusion of debugging symbol table information in the compiled object module. The default is /DEBUG or, equivalently, /DEBUG=ALL.

- /DESIGN or /NODESIGN

  Controls whether the input file is processed as a design or compiled as an Ada source. The default is /NODESIGN, in

which case the file is compiled.

- /DIAGNOSTICS /DIAGNOSTICS=file-name, or /NODIAGNOSTICS

  Controls whether a special diagnostics file is produced for use with the VAX Language-Sensitive Editor (a separate DIGITAL product). The default is /NODIAGNOSTICS.

- /ERROR_LIMIT=n

  Controls the number of error level diagnostics that are allowed within a single compilation unit before the compilation is aborted. The default is /ERROR_LIMIT=30.

- /LIBRARY=directory-name

  Specifies the name of the Ada compilation library to be used as the context for the compilation. The default is the library last established by the ACS SET LIBRARY command.

- /LIST, /LIST=file-name, or /NOLIST

  Controls whether a listing file is produced. /LIST without a file-name uses a default file-name of the form sourcename.LIS, where sourcename is the name of the source file being compiled. The default is /NOLIST (for both interactive and batch mode).

- /LOAD or /NOLOAD

  Controls whether the current program library is updated with successfully processed units contained in the source file. The default is /LOAD,

- /MACHINE_CODE or /NOMACHINE_CODE

  Controls whether generated machine code (approximating assembler notation) is included in the listing file, if produced. The default is /NOMACHINE_CODE.

- /NOTE_SOURCE or /NONOTE_SOURCE

  Controls whether the file specification of the current source file is noted in the compilation library. (This copy is used for certain automated (re)compilation features.) The default is /NOTE_SOURCE.

- /OPTIMIZE or /NOOPTIMIZE

  Controls whether full or minimal optimization is applied in producing the compiled code. The default is /OPTIMIZE. (/NOOPTIMIZE is primarily of use in combination with /DEBUG.)

- /SYNTAX_ONLY or /NOSYNTAX_ONLY

  Controls whether a syntax check only is performed. The default is /NOSYNTAX_ONLY, which indicates that full processing is done.

- /SHOW=PORTABILITY or /NOSHOW

  Controls whether a portability summary is included in the listing. The default is /SHOW=PORTABILITY.

- /WARNINGS=(category:destination,...)

  Specifies which categories of informational and warning level messages are displayed for which destinations. The categories can be WARNINGS, WEAK_WARNINGS, SUPPLEMENTAL, COMPILATION_NOTES and STATUS. The destinations can be ALL, NONE or combinations of TERMINAL, LISTING or DIAGNOSTICS. The default is

  /WARNINGS=(WARN:ALL,WEAK:ALL,SUPP:ALL,COMP:NONE,STAT:LIST)

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

Linker options for DEC Ada hosted on OpenVMS VAX and OpenVMS AXP systems

DEC Ada programs are linked using the "ACS LINK" command. The default ACS LINK options were used except as follows:

1. /COMMAND was used. ACS LINK checks that all units in a program are current and writes a command file to create the executable image using the VMS linker. By default, this command file is invoked in a subprocess and the image is linked. If /COMMAND is specified, the command file is written but not invoked. For validation, /COMMAND was used, and the generated command file was invoked after the ACS LINK command to do the actual link. This approach saves the overhead of spawning a subprocess for each executable validation test.

2. /NOTRACE was specified to exclude traceback symbol information from the image file since this information is only relevant when debugging programs.

3. /EXECUTABLE was used to specify the name of the executable image file.

4. For VAXELN targets only, /SYSTEM_NAME=VAXELN was used.

The options and defaults for linking Ada programs are summarized below.

- /COMMAND=file-name

  Write a command file create the executable image using the VMS linker but do not invoke this commmand file.

- /DEBUG or /NODEBUG

  Controls whether a debugger symbol table is included in the executable image. The default is /NODEBUG.

- /EXECUTABLE, /EXECUTABLE=file-name, or /NOEXECUTABLE

  Controls whether the linker creates an executable image file and optionally provides the name of the file. The default is /EXECUTABLE.

- /LOG or /NOLOG

  Controls wheter a list of all units in the image is displayed. The default is /NOLOG.

- /MAIN or /NOMAIN

  Specifies whether the main Ada unit is the main program. The

default is /MAIN which indicates that the main Ada unit is the main program.

o /MAP, /MAP=file-name, or /NOMAP

  Controls whether the linker creates an image map listing file. The default is /NOMAP. If /MAP is specified, some other options can be specified to control the level of detail in the map listing file.

o /OUTPUT=file-name

  If specified, requests that output be written to a file other than to the standard output device.

o /SYSLIB or /NOSYSLIB

  Controls whether the linker automatically searches the default system library for unresolved references. By default, it is automatically searched.

o /SYSSHR or /NOSYSSHR

  Controls whether the linker automatically searches the default system shareable image library for unresolved references. By default, it is automatically searched.

o /SYSTEM_NAME=system

  Directs the program library manager to produce an image for execution on a particular operating system. On VAX systems, the possible values are VAX_VMS or VAXELN. On AXP systems, the only value supported is OpenVMS_AXP.

  If /SYSTEM_NAME is not specified, the setting of the pragma SYSTEM_NAME determines the target environment.

o /TRACEBACK or /NOTRACEBACK

  Controls whether subprogram traceback information is included in the executable image for run-time error reporting. The default is /TRACEBACK.

o /USERLIBRARY=(table,...) or /NOUSERLIBRARY

  Controls whether the linker searches any user-defined default libraries for unresolved external symbols. By default, the linker searches process, group, and system logical name tables for user-defined library definitions.

Additional options are provided that control whether the link is done while the user waits or is done in a background mode. Using one option or the other has no effect on the executable image that is generated.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

type INTEGER is range -2147483648..2147483647;
type LONG_INTEGER is range -2147483648..2147483647;
type SHORT_INTEGER is range -32768..32768;
type SHORT_SHORT_INTEGER is range -128..127;


type FLOAT is digits 6 range -1.70141E+38..1.70141E+38;
type LONG_FLOAT is digits 15 range
             -8.988465674312E+307..8.988465674312E+307;
type LONG_LONG_FLOAT is digits 33 range
             -5.948657476786158825428796633140OE+4931..
              5.948657476786158825428796633140OE+4931;

type DURATION is delta 1.0E-4 range -131072.0..131071.9999;

end STANDARD;
```

B
Predefined Language Pragmas

1 This annex defines the pragmas LIST, PAGE, and OPTIMIZE, and summarizes the definitions given elsewhere of the remaining language-defined pragmas.

The DEC Ada pragmas IDENT and TITLE are also defined in this annex.

Pragma	Meaning
AST_ENTRY	On VMS systems only. Takes the simple name of a single entry as the single argument; at most one AST_ENTRY pragma is allowed for any given entry. This pragma must be used in combination with the AST_ENTRY attribute, and is only allowed after the entry declaration and in the same task type specification or single task as the entry to which it applies. This pragma specifies that the given entry may be used to handle a VMS asynchronous system trap (AST) resulting from a VMS system service call. The pragma does not affect normal use of the entry (see 9.12a).
COMMON_OBJECT	Takes an internal name denoting an object, and optionally takes an external designator (the name of a linker storage area) and a size as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a variable declared by an earlier declarative item of the same declarative part or

package specification. The variable must have a size that is known at compile time, and it must not require implicit initialization. This pragma is not allowed for objects declared with a renaming declaration. This pragma enables the shared use of objects that are stored in overlaid storage areas (see 13.9a.2.3).

COMPONENT_ALIGNMENT

Takes an alignment choice and optionally the simple name of an array or record type as arguments. When no simple name is specified, the pragma must occur within a declarative part or package specification, and the effect of the pragma extends to types declared from the place of the pragma to the end of the innermost declarative part or package specification in which the pragma was declared. When a simple name is specified, the pragma and the type declaration must both occur immediately within the same declarative part, package specification, or task specification; the declaration must occur before the pragma. The position of the pragma and the restrictions on the named type are governed by the same rules as those for a representation clause. This pragma specifies the kind of alignment used for the components of the array or record types to which it applies (see 13.1a).

2 CONTROLLED

Takes the simple name of an access type as the single argument. This pragma is only allowed immediately within the declarative part or package specification that contains the declaration of the access type; the declaration must occur before the pragma. This pragma is not allowed for a derived type. This pragma specifies

that automatic storage reclamation must not be performed for objects designated by values of the access type, except upon leaving the innermost block statement, subprogram body, or task body that encloses the access type declaration, or after leaving the main program (see 4.8).

3 ELABORATE

Takes one or more simple names denoting library units as arguments. This pragma is only allowed immediately after the context clause of a compilation unit (before the subsequent library unit or secondary unit). Each argument must be the simple name of a library unit mentioned by the context clause. This pragma specifies that the corresponding library unit body must be elaborated before the given compilation unit. If the given compilation unit is a subunit, the library unit body must be elaborated before the body of the ancestor library unit of the subunit (see 10.5).

EXPORT_EXCEPTION

On VMS systems only.

Takes an internal name denoting an exception, and optionally takes an external designator (the name of a linker global symbol), a form (ADA or VMS), and a code (a static integer expression that is interpreted as a condition code) as arguments. A code value must be specified when the form is VMS (the default if the form is not specified). This pragma is only allowed at the place of a declarative item, and must apply to an exception declared by an earlier declarative item of the same declarative part or package specification; it is not

allowed for an exception declared with a renaming declaration or for an exception declared in a generic unit. This pragma permits an Ada exception to be handled by programs written in another programming language (see 13.9a.3.2).

EXPORT_FUNCTION

Takes an internal name denoting a function, and optionally takes an external designator (the name of a linker global symbol), parameter types, result type, parameter mechanisms, and result mechanism as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a function declared by an earlier declarative item of the same declarative part or package specification. In the case of a function declared as a compilation unit, the pragma is only allowed after the function declaration and before any subsequent compilation unit. This pragma is not allowed for a function declared with a renaming declaration, and it is not allowed for a generic function (it may be given for a generic instantiation). This pragma permits an Ada function to be called from a program written in another programming language (see 13.9a.1.3).

EXPORT_OBJECT

Takes an internal name denoting an object, and optionally takes an external designator (the name of a linker global symbol) and size option (a linker absolute global symbol that will be defined in the object module—useful on VMS systems only) as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a constant or a variable declared by an earlier declarative

	item of the same declarative part or package specification; the declaration must occur at the outermost level of a library package specification or body. The object to be exported must have a size that is known at compile time. This pragma is not allowed for objects declared with a renaming declaration, and is not allowed in a generic unit. This pragma permits an Ada object to be referred to by a routine written in another programming language (see 13.9a.2.2).
EXPORT_PROCEDURE	Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a linker global symbol), parameter types, and parameter mechanisms as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. This pragma is not allowed for a procedure declared with a renaming declaration, and is not allowed for a generic procedure (it may be given for a generic instantiation). This pragma permits an Ada routine to be called from a program written in another programming language (see 13.9a.1.3).
EXPORT_VALUED_PROCEDURE	Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a linker global symbol), parameter types, and parameter mechanisms as arguments. This pragma is only

	allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. The first (or only) parameter of the procedure must be of mode **out**. This pragma is not allowed for a procedure declared with a renaming declaration and is not allowed for a generic procedure (it may be given for a generic instantiation). This pragma permits an Ada procedure to behave as a function that both returns a value and causes side effects on its parameters when it is called from a routine written in another programming language (see 13.9a.1.3).
FLOAT_REPRESENTATION	On VMS systems only. On VMS VAX systems, takes VAX_FLOAT as the single argument. On VMS AXP systems, takes either VAX_FLOAT or IEEE_FLOAT as the single argument; the default is VAX_FLOAT. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. It specifies the choice of representation to be used for the predefined floating point types in the packages SYSTEM and STANDARD. (see 3.5.7a).
IDENT	Takes a string literal of 31 or fewer characters as the single argument. The pragma IDENT has the following form: `pragma IDENT (string_literal);`

This pragma is allowed only in the outermost declarative part or declarative items of a compilation unit. The given string is used to identify the object module associated with the compilation unit in which the pragma IDENT occurs.

IMPORT_EXCEPTION	On VMS systems only. Takes an internal name denoting an exception, and optionally takes an external designator (the name of a linker global symbol), a form (ADA or VMS), and a code (a static integer expression that is interpreted as a condition code) as arguments. A code value is allowed only when the form is VMS (the default if the form is not specified). This pragma is only allowed at the place of a declarative item, and must apply to an exception declared by an earlier declarative item of the same declarative part or package specification; it is not allowed for an exception declared with a renaming declaration. This pragma permits a non-Ada exception (most notably, a VMS condition) to be handled by an Ada program (see 13.9a.3.1).
IMPORT_FUNCTION	Takes an internal name denoting a function, and optionally takes an external designator (the name of a linker global symbol), parameter types, result type, parameter mechanisms, and result mechanism as arguments. On VMS systems, a first optional parameter is also available as an argument. The pragma INTERFACE must be used with this pragma (see 13.9). This pragma is only allowed at the place of a declarative item, and must apply to a function declared

by an earlier declarative item of the same declarative part or package specification. In the case of a function declared as a compilation unit, the pragma is only allowed after the function declaration and before any subsequent compilation unit. This pragma is allowed for a function declared with a renaming declaration; it is not allowed for a generic function or a generic function instantiation. This pragma permits a non-Ada routine to be used as an Ada function (see 13.9a.1.1).

IMPORT_OBJECT	Takes an internal name denoting an object, and optionally takes an external designator (the name of a linker global symbol) and size (a linker absolute global symbol that will be defined in the object module—useful on VMS systems only) as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a variable declared by an earlier declarative item of the same declarative part or package specification. The variable must have a size that is known at compile time, and it cannot have an initial value. This pragma is not allowed for objects declared with a renaming declaration. This pragma permits storage declared in a non-Ada routine to be referred to by an Ada program (see 13.9a.2.1).
IMPORT_PROCEDURE	Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a linker global symbol), parameter types, and parameter mechanisms as arguments. On VMS systems, a first optional parameter is also available as an argument. The pragma

INTERFACE must be used with this pragma (see 13.9). This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. This pragma is allowed for a procedure declared with a renaming declaration; it is not allowed for a generic procedure or a generic procedure instantiation. This pragma permits a non-Ada routine to be used as an Ada procedure (see 13.9a.1.1).

IMPORT_VALUED_PROCEDURE Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a linker global symbol), parameter types, and parameter mechanisms as arguments. On VMS systems, a first optional parameter is also available as an argument. The pragma INTERFACE must be used with this pragma (see 13.9). This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. The first (or only) parameter of the procedure must be of mode **out**. This pragma is allowed for a procedure declared with a renaming declaration; it is not allowed for a generic procedure. This

pragma permits a non-Ada routine that returns a value and causes side effects on its parameters to be used as an Ada procedure (see 13.9a.1.1).

4 INLINE

Takes one or more names as arguments; each name is either the name of a subprogram or the name of a generic subprogram. This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit. This pragma specifies that the subprogram bodies should be expanded inline at each call whenever possible; in the case of a generic subprogram, the pragma applies to calls of its instantiations (see 6.3.2).

INLINE_GENERIC

Takes one or more names as arguments; each name is either the name of a generic declaration or the name of an instance of a generic declaration. This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit. Each argument must be the simple name of a generic subprogram or package, or a (nongeneric) subprogram or package that is an instance of a generic subprogram or package declared by an earlier declarative item of the same declarative part or package specification. This pragma specifies that inline expansion of the generic body is desired for each instantiation of the named generic declarations or of the particular named instances; the pragma does not apply to calls of

instances of generic subprograms (see 12.1a).

5 INTERFACE

Takes a language name and a subprogram name as arguments. This pragma is allowed at the place of a declarative item, and must apply in this case to a subprogram declared by an earlier declarative item of the same declarative part or package specification. This pragma is also allowed for a library unit; in this case the pragma must appear after the subprogram declaration, and before any subsequent compilation unit. This pragma specifies the other language (and thereby the calling conventions) and informs the compiler that an object module will be supplied for the corresponding subprogram (see 13.9).

In DEC Ada, the pragma INTERFACE is required in combination with the pragmas IMPORT_FUNCTION, IMPORT_PROCEDURE, and IMPORT_VALUED_PROCEDURE when any of those pragmas are used (see 13.9a.1).

INTERFACE_NAME

Takes an internal name and an external name as arguments. The internal name may be an Ada simple name that denotes a subprogram or an object. If the declared entity is a function, the internal name may be a string literal that denotes an operator symbol. The external name may be any string literal; the literal is used as a linker global symbol that is associated with the external subprogram or object. This pragma is only allowed at the place of a declarative item, and must apply to an entity declared by an earlier declarative item of the

same declarative part or package specification.

If this pragma applies to a subprogram, then the pragma INTERFACE must also apply (see 13.9). If a subprogram has been declared as a compilation unit, the pragma is only allowed after the subprogram declaration and before any subsequent compilation unit. This pragma is allowed for subprograms declared with a renaming declaration. This pragma is not allowed for a generic subprogram or a generic subprogram instantiation.

If this pragma applies to an object, then the size of the object must be known at compile time. This pragma is not allowed for an the object declared with a renaming declaration.

This pragma associates an external symbol with the internal Ada name for a subprogram or object (see 13.9b).

6	LIST	Takes one of the identifiers ON or OFF as the single argument. This pragma is allowed anywhere a pragma is allowed. It specifies that listing of the compilation is to be continued or suspended until a LIST pragma with the opposite argument is given within the same compilation. The pragma itself is always listed if the compiler is producing a listing.
	LONG_FLOAT	On VMS systems only. Also, the value of the pragma FLOAT_ REPRESENTATION must be VAX_ FLOAT.

Takes either D_FLOAT or G_FLOAT as the single argument. The default is G_FLOAT. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. It specifies the choice of representation to be used for the predefined type LONG_FLOAT in the package STANDARD, and for floating point type declarations with **digits** specified in the range 7 .. 15 (see 3.5.7b).

MAIN_STORAGE

On VMS VAX systems only.

Takes one or two nonnegative static simple expressions of some integer type as arguments. This pragma is only allowed in the outermost declarative part of a library subprogram; at most one such pragma is allowed in a library subprogram. It has an effect only when the subprogram to which it applies is used as a main program. This pragma causes a fixed-size stack to be created for a main task (the task associated with a main program), and determines the number of storage units (bytes) to be allocated for the stack working storage area or guard pages or both. The value specified for either or both the working storage area and guard pages is rounded up to an integral number of pages. A value of zero for the working storage area results in the use of a default size; a value of zero for the guard pages results in no guard storage. A negative value for either working storage or guard pages causes the pragma to be ignored (see 13.2b).

7 MEMORY_SIZE

Takes a numeric literal as the single argument. This pragma is only allowed at the start of a compilation, before the

first compilation unit (if any) of the compilation. The effect of this pragma is to use the value of the specified numeric literal for the definition of the named number MEMORY_SIZE (see 13.7).

8 OPTIMIZE

Takes one of the identifiers TIME or SPACE as the single argument. This pragma is only allowed within a declarative part and it applies to the block or body enclosing the declarative part. It specifies whether time or space is the primary optimization criterion.

In DEC Ada, this pragma is only allowed immediately within a declarative part of a body declaration.

9 PACK

Takes the simple name of a record or array type as the single argument. The allowed positions for this pragma, and the restrictions on the named type, are governed by the same rules as for a representation clause. The pragma specifies that storage minimization should be the main criterion when selecting the representation of the given type (see 13.1).

10 PAGE

This pragma has no argument, and is allowed anywhere a pragma is allowed. It specifies that the program text which follows the pragma should start on a new page (if the compiler is currently producing a listing).

11 PRIORITY

Takes a static expression of the predefined integer subtype PRIORITY as the single argument. This pragma is only allowed within the specification of a task unit or immediately within the outermost declarative part of a main program. It specifies the priority of the

task (or tasks of the task type) or the priority of the main program (see 9.8).

PSECT_OBJECT

On VMS systems only.

Has the same syntax and the same effect as the pragma COMMON_ OBJECT (see 13.9a.2.3).

12 SHARED

Takes the simple name of a variable as the single argument. This pragma is allowed only for a variable declared by an object declaration and whose type is a scalar or access type; the variable declaration and the pragma must both occur (in this order) immediately within the same declarative part or package specification. This pragma specifies that every read or update of the variable is a synchronization point for that variable. An implementation must restrict the objects for which this pragma is allowed to objects for which each of direct reading and direct updating is implemented as an indivisible operation (see 9.11).

SHARE_GENERIC

On VMS systems only.

Takes one or more names as arguments; each name is either the name of a generic declaration or the name of an instance of a generic declaration. This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit. Each argument either must be the simple name of a generic subprogram or package, or it must be a (nongeneric) subprogram or package that is an instance of a generic subprogram or package. If the argument is an instance of a generic

subprogram or package, then it must be declared by an earlier declarative item of the same declarative part or package specification. This pragma specifies that generic code sharing is desired for each instantiation of the named generic declarations or of the particular named instances (see 12.1b).

13 STORAGE_UNIT

Takes a numeric literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. The effect of this pragma is to use the value of the specified numeric literal for the definition of the named number STORAGE_UNIT (see 13.7).

In DEC Ada, the only argument allowed for this pragma is 8 (bits).

14 SUPPRESS

Takes as arguments the identifier of a check and optionally also the name of either an object, a type or subtype, a subprogram, a task unit, or a generic unit. This pragma is only allowed either immediately within a declarative part or immediately within a package specification. In the latter case, the only allowed form is with a name that denotes an entity (or several overloaded subprograms) declared immediately within the package specification. The permission to omit the given check extends from the place of the pragma to the end of the declarative region associated with the innermost enclosing block statement or program unit. For a pragma given in a package specification, the permission extends to the end of the scope of the named entity.

If the pragma includes a name, the permission to omit the given check is further restricted: it is given only for operations on the named object or on all objects of the base type of a named type or subtype; for calls of a named subprogram; for activations of tasks of the named task type; or for instantiations of the given generic unit (see 11.7).

SUPPRESS_ALL

This pragma has no argument and is only allowed following a compilation unit. This pragma specifies that all run-time checks in the unit are suppressed (see 11.7).

15 SYSTEM_NAME

Takes an enumeration literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. The effect of this pragma is to use the enumeration literal with the specified identifier for the definition of the constant SYSTEM_NAME. This pragma is only allowed if the specified identifier corresponds to one of the literals of the type NAME declared in the package SYSTEM (see 13.7).

TASK_STORAGE

Takes the simple name of a task type and a static expression of some integer type as arguments. This pragma is allowed anywhere that a task storage specification is allowed; that is, the declaration of the task type to which the pragma applies and the pragma must both occur (in this order) immediately within the same declarative part, package specification, or task specification. The effect of this pragma is to use the value of the expression as the number of storage

units (bytes) to be allocated as guard storage. The value is rounded up to an appropriate boundary. A negative value causes the pragma to be ignored. A zero value has system-specific results: on VMS VAX systems, a value of zero results in no guard storage; on VMS AXP and ULTRIX systems, a value of zero results in a minimal guard area (see 13.2a).

TIME_SLICE

On VMS systems only.

Takes a static expression of the predefined fixed point type DURATION (*in the package* STANDARD) as the single argument. This pragma is only allowed in the outermost declarative part of a library subprogram, and at most one such pragma is allowed in a library subprogram. It has an effect only when the subprogram to which it applies is used as a main program. This pragma causes the task scheduler to turn time slicing on or off and, on some systems, to limit the amount of continuous execution time given to a task
(see 9.8a; see also the appropriate run-time reference manual for implementation differences across systems).

TITLE

Takes a title or a subtitle string, or both, as arguments. The pragma TITLE has the following form:

```
pragma TITLE (titling-option
   [,titling-option]);

titling-option :=
      [TITLE =>] string_literal
   | [SUBTITLE =>] string_literal
```

This pragma is allowed anywhere a pragma is allowed; the given strings supersede the default title and/or subtitle portions of a compilation listing.

VOLATILE

Takes the simple name of a variable as the single argument. This pragma is only allowed for a variable declared by an object declaration. The variable declaration and the pragma must both occur (in this order) immediately within the same declarative part or package specification. The pragma must appear before any occurrence of the name of the variable other than in an address clause or in one of the DEC Ada pragmas IMPORT_OBJECT, EXPORT_OBJECT, COMMON_OBJECT, or PSECT_OBJECT. The variable cannot be declared by a renaming declaration. The pragma VOLATILE specifies that the variable may be modified asynchronously. This pragma instructs the compiler to obtain the value of a variable from memory each time it is used (see 9.11).

F

Implementation-Dependent Characteristics

Note

This appendix is not part of the standard definition of the Ada programming language.

This appendix summarizes the implementation-dependent characteristics of DEC Ada by presenting the following:

- Lists of the DEC Ada pragmas and attributes.
- The specifications of the package SYSTEM.
- The restrictions on representation clauses and unchecked type conversions.
- The conventions for names denoting implementation-dependent components in record representation clauses.
- The interpretation of expressions in address clauses.
- The implementation-dependent characteristics of the input-output packages.
- Other implementation-dependent characteristics.

See the relevant run-time reference manual for additional implementation-specific details.

F.1 Implementation-Dependent Pragmas

DEC Ada provides the following pragmas, which are defined elsewhere in the text. In addition, DEC Ada restricts the predefined language pragmas INLINE and INTERFACE. See Annex B for a descriptive pragma summary.

Attribute	DEC Ada systems on which it applies	Section
AST_ENTRY	OpenVMS	9.12a
BIT	All	13.7.2
MACHINE_SIZE	All	13.7.2
NULL_PARAMETER	All	13.9a.1.2
TYPE_CLASS	All	13.7a.2

F.3 Specification of the Package System

DEC Ada provides a system-specific version of the package SYSTEM for each system on which it is supported. The individual package SYSTEM specifications appear in the following sections.

F.3.1 The Package System on OpenVMS VAX Systems

```
package SYSTEM is

   type NAME is
      -- DEC Ada implementations
     (VAX_VMS, VAXELN, OpenVMS_AXP, RISC_ULTRIX,
      -- XD Ada implementations
      MIL_STD_1750A, MC68000, MC68020, MC68040, CPU32);
   for NAME use (1, 2, 7, 8, 101, 102, 103, 104, 105);

   SYSTEM_NAME   : constant NAME := VAX_VMS;
   STORAGE_UNIT  : constant := 8;
   MEMORY_SIZE   : constant := 2**31-1;
   MAX_INT       : constant := 2**31-1;
   MIN_INT       : constant := -(2**31);
   MAX_DIGITS    : constant := 33;
   MAX_MANTISSA  : constant := 31;
   FINE_DELTA    : constant := 2.0**(-31);
   TICK          : constant := 10.0**(-2);

   subtype PRIORITY is INTEGER range 0 .. 15;

   type INTEGER_8 is range -128 .. 127;
   for INTEGER_8'SIZE use 8;

   type INTEGER_16 is range -32_768 .. 32_767;
   for INTEGER_16'SIZE use 16;

   type INTEGER_32 is range -2_147_483_648 .. 2_147_483_647;
   for INTEGER_32'SIZE use 32;

   type LARGEST_INTEGER is range MIN_INT .. MAX_INT;
```

```
-- Address type
--
   type ADDRESS is private;

   ADDRESS_ZERO : constant ADDRESS;
   NO_ADDR      : constant ADDRESS;
   NULL_ADDRESS : constant ADDRESS;

-- Note that because ADDRESS is a private type
-- the functions "=" and "/=" are already available and
-- do not have to be explicitly defined
--
-- function "="  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
   function "<"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
   function "<=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
   function ">"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
   function ">=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;

   generic
      type TARGET is private;
   function FETCH_FROM_ADDRESS (A : ADDRESS) return TARGET;

   generic
      type TARGET is private;
   procedure ASSIGN_TO_ADDRESS (A : ADDRESS; T : TARGET);

-- DEC Ada floating point type declarations for the VAX
-- floating point data types

   type F_FLOAT is (digits 6);
   type D_FLOAT is (digits 9);
   type G_FLOAT is (digits 15);
   type H_FLOAT is (digits 33);

   type TYPE_CLASS is (TYPE_CLASS_ENUMERATION,
                       TYPE_CLASS_INTEGER,
                       TYPE_CLASS_FIXED_POINT,
                       TYPE_CLASS_FLOATING_POINT,
                       TYPE_CLASS_ARRAY,
                       TYPE_CLASS_RECORD,
                       TYPE_CLASS_ACCESS,
                       TYPE_CLASS_TASK,
                       TYPE_CLASS_ADDRESS);

-- AST handler type

   type AST_HANDLER is limited private;

   NO_AST_HANDLER : constant AST_HANDLER;

-- Non-Ada exception

   NON_ADA_ERROR : exception;

-- Hardware-oriented types and functions
```

```
type BIT_ARRAY is array (INTEGER range <>) of BOOLEAN;
pragma PACK(BIT_ARRAY);

subtype BIT_ARRAY_8  is BIT_ARRAY (0 ..  7);
subtype BIT_ARRAY_16 is BIT_ARRAY (0 .. 15);
subtype BIT_ARRAY_32 is BIT_ARRAY (0 .. 31);
subtype BIT_ARRAY_64 is BIT_ARRAY (0 .. 63);

type UNSIGNED_BYTE is range 0 .. 255;
for  UNSIGNED_BYTE'SIZE use 8;

function "not" (LEFT        : UNSIGNED_BYTE) return UNSIGNED_BYTE;
function "and" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
function "or"  (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
function "xor" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;

function TO_UNSIGNED_BYTE (X : BIT_ARRAY_8) return UNSIGNED_BYTE;
function TO_BIT_ARRAY_8   (X : UNSIGNED_BYTE) return BIT_ARRAY_8;

type UNSIGNED_BYTE_ARRAY is array (INTEGER range <>) of UNSIGNED_BYTE;

type UNSIGNED_WORD is range 0 .. 65535;
for  UNSIGNED_WORD'SIZE use 16;

function "not" (LEFT        : UNSIGNED_WORD) return UNSIGNED_WORD;
function "and" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
function "or"  (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
function "xor" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;

function TO_UNSIGNED_WORD (X : BIT_ARRAY_16) return UNSIGNED_WORD;
function TO_BIT_ARRAY_16  (X : UNSIGNED_WORD) return BIT_ARRAY_16;

type UNSIGNED_WORD_ARRAY is array (INTEGER range <>) of UNSIGNED_WORD;

type UNSIGNED_LONGWORD is range -2_147_483_648 .. 2_147_483_647;
for  UNSIGNED_LONGWORD'SIZE use 32;

function "not" (LEFT        : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "and" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "or"  (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "xor" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;

function TO_UNSIGNED_LONGWORD (X : BIT_ARRAY_32) return UNSIGNED_LONGWORD;
function TO_BIT_ARRAY_32 (X : UNSIGNED_LONGWORD) return BIT_ARRAY_32;

type UNSIGNED_LONGWORD_ARRAY is
   array (INTEGER range <>) of UNSIGNED_LONGWORD;
```

```
type UNSIGNED_QUADWORD is
   record
      L0 : UNSIGNED_LONGWORD;
      L1 : UNSIGNED_LONGWORD;
   end record;
for  UNSIGNED_QUADWORD'SIZE use 64;
for UNSIGNED_QUADWORD use
   record
      at mod 8;
   end record;

function "not" (LEFT        : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
function "and" (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
function "or"  (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
function "xor" (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;

function TO_UNSIGNED_QUADWORD (X : BIT_ARRAY_64) return UNSIGNED_QUADWORD;
function TO_BIT_ARRAY_64 (X : UNSIGNED_QUADWORD) return BIT_ARRAY_64;

type UNSIGNED_QUADWORD_ARRAY is
   array (INTEGER range <>) of UNSIGNED_QUADWORD;

function TO_ADDRESS  (X : INTEGER)               return ADDRESS;
function TO_ADDRESS  (X : UNSIGNED_LONGWORD)     return ADDRESS;
function TO_ADDRESS  (X : {universal_integer})   return ADDRESS;

function TO_INTEGER           (X : ADDRESS)     return INTEGER;
function TO_UNSIGNED_LONGWORD (X : ADDRESS)     return UNSIGNED_LONGWORD;

function TO_UNSIGNED_LONGWORD (X : AST_HANDLER) return UNSIGNED_LONGWORD;

-- Conventional names for static subtypes of type UNSIGNED_LONGWORD
```

```
   subtype UNSIGNED_1  is UNSIGNED_LONGWORD range 0 .. 2** 1-1;
   subtype UNSIGNED_2  is UNSIGNED_LONGWORD range 0 .. 2** 2-1;
   subtype UNSIGNED_3  is UNSIGNED_LONGWORD range 0 .. 2** 3-1;
   subtype UNSIGNED_4  is UNSIGNED_LONGWORD range 0 .. 2** 4-1;
   subtype UNSIGNED_5  is UNSIGNED_LONGWORD range 0 .. 2** 5-1;
   subtype UNSIGNED_6  is UNSIGNED_LONGWORD range 0 .. 2** 6-1;
   subtype UNSIGNED_7  is UNSIGNED_LONGWORD range 0 .. 2** 7-1;
   subtype UNSIGNED_8  is UNSIGNED_LONGWORD range 0 .. 2** 8-1;
   subtype UNSIGNED_9  is UNSIGNED_LONGWORD range 0 .. 2** 9-1;
   subtype UNSIGNED_10 is UNSIGNED_LONGWORD range 0 .. 2**10-1;
   subtype UNSIGNED_11 is UNSIGNED_LONGWORD range 0 .. 2**11-1;
   subtype UNSIGNED_12 is UNSIGNED_LONGWORD range 0 .. 2**12-1;
   subtype UNSIGNED_13 is UNSIGNED_LONGWORD range 0 .. 2**13-1;
   subtype UNSIGNED_14 is UNSIGNED_LONGWORD range 0 .. 2**14-1;
   subtype UNSIGNED_15 is UNSIGNED_LONGWORD range 0 .. 2**15-1;
   subtype UNSIGNED_16 is UNSIGNED_LONGWORD range 0 .. 2**16-1;
   subtype UNSIGNED_17 is UNSIGNED_LONGWORD range 0 .. 2**17-1;
   subtype UNSIGNED_18 is UNSIGNED_LONGWORD range 0 .. 2**18-1;
   subtype UNSIGNED_19 is UNSIGNED_LONGWORD range 0 .. 2**19-1;
   subtype UNSIGNED_20 is UNSIGNED_LONGWORD range 0 .. 2**20-1;
   subtype UNSIGNED_21 is UNSIGNED_LONGWORD range 0 .. 2**21-1;
   subtype UNSIGNED_22 is UNSIGNED_LONGWORD range 0 .. 2**22-1;
   subtype UNSIGNED_23 is UNSIGNED_LONGWORD range 0 .. 2**23-1;
   subtype UNSIGNED_24 is UNSIGNED_LONGWORD range 0 .. 2**24-1;
   subtype UNSIGNED_25 is UNSIGNED_LONGWORD range 0 .. 2**25-1;
   subtype UNSIGNED_26 is UNSIGNED_LONGWORD range 0 .. 2**26-1;
   subtype UNSIGNED_27 is UNSIGNED_LONGWORD range 0 .. 2**27-1;
   subtype UNSIGNED_28 is UNSIGNED_LONGWORD range 0 .. 2**28-1;
   subtype UNSIGNED_29 is UNSIGNED_LONGWORD range 0 .. 2**29-1;
   subtype UNSIGNED_30 is UNSIGNED_LONGWORD range 0 .. 2**30-1;
   subtype UNSIGNED_31 is UNSIGNED_LONGWORD range 0 .. 2**31-1;

-- Functions for obtaining global symbol values

   function IMPORT_VALUE (SYMBOL : STRING) return UNSIGNED_LONGWORD;
   function IMPORT_ADDRESS (SYMBOL : STRING) return ADDRESS;

-- VAX device and process register operations

   function  READ_REGISTER (SOURCE : UNSIGNED_BYTE) return UNSIGNED_BYTE;
   function  READ_REGISTER (SOURCE : UNSIGNED_WORD) return UNSIGNED_WORD;
   function  READ_REGISTER (SOURCE : UNSIGNED_LONGWORD)
      return UNSIGNED_LONGWORD;

   procedure WRITE_REGISTER(SOURCE : UNSIGNED_BYTE;
                            TARGET : out UNSIGNED_BYTE);
   procedure WRITE_REGISTER(SOURCE : UNSIGNED_WORD;
                            TARGET : out UNSIGNED_WORD);
   procedure WRITE_REGISTER(SOURCE : UNSIGNED_LONGWORD;
                            TARGET : out UNSIGNED_LONGWORD);

   function  MFPR (REG_NUMBER : INTEGER) return UNSIGNED_LONGWORD;
   procedure MTPR (REG_NUMBER : INTEGER;
                   SOURCE     : UNSIGNED_LONGWORD);
```

```
-- VAX interlocked-instruction procedures

  procedure CLEAR_INTERLOCKED (BIT       : in out BOOLEAN;
                               OLD_VALUE : out BOOLEAN);
  procedure SET_INTERLOCKED   (BIT       : in out BOOLEAN;
                               OLD_VALUE : out BOOLEAN);

  type ALIGNED_WORD is
     record
        VALUE : SHORT_INTEGER;
     end record;
  for ALIGNED_WORD use
     record
        at mod 2;
     end record;

  procedure ADD_INTERLOCKED (ADDEND : in     SHORT_INTEGER;
                             AUGEND : in out ALIGNED_WORD;
                             SIGN   : out    INTEGER);

  type INSQ_STATUS is (OK_NOT_FIRST, FAIL_NO_LOCK, OK_FIRST);
  for INSQ_STATUS use (0, 1, 2);

  type REMQ_STATUS is (OK_NOT_EMPTY, FAIL_NO_LOCK,
                       OK_EMPTY,     FAIL_WAS_EMPTY);
  for REMQ_STATUS use (0, 1, 2, 3);

  procedure INSQHI (ITEM   : in  ADDRESS;
                    HEADER : in  ADDRESS;
                    STATUS : out INSQ_STATUS);

  procedure REMQHI (HEADER : in  ADDRESS;
                    ITEM   : out ADDRESS;
                    STATUS : out REMQ_STATUS);

  procedure INSQTI (ITEM   : in  ADDRESS;
                    HEADER : in  ADDRESS;
                    STATUS : out INSQ_STATUS);

  procedure REMQTI (HEADER : in  ADDRESS;
                    ITEM   : out ADDRESS;
                    STATUS : out REMQ_STATUS);

private

   -- Not shown

end SYSTEM;
```

F.3.2 The Package System on OpenVMS AXP Systems

```
package SYSTEM is

   type NAME is
       -- DEC Ada implementations
      (VAX_VMS, VAXELN, OpenVMS_AXP, RISC_ULTRIX,
       -- XD Ada implementations
       MIL_STD_1750A, MC68000, MC68020, MC68040, CPU32);
   for NAME use (1, 2, 7, 8, 101, 102, 103, 104, 105);

   SYSTEM_NAME   : constant NAME := OpenVMS_AXP;
   STORAGE_UNIT  : constant := 8;
   MEMORY_SIZE   : constant := 2**31-1;
   MAX_INT       : constant := 2**31-1;
   MIN_INT       : constant := -(2**31);
   MAX_DIGITS    : constant := 15;
   MAX_MANTISSA  : constant := 31;
   FINE_DELTA    : constant := 2.0**(-31);
   TICK          : constant := 10.0**(-3);

   subtype PRIORITY is INTEGER range 0 .. 15;

   type INTEGER_8 is range -128 .. 127;
   for INTEGER_8'SIZE use 8;

   type INTEGER_16 is range -32_768 .. 32_767;
   for INTEGER_16'SIZE use 16;

   type INTEGER_32 is range -2_147_483_648 .. 2_147_483_647;
   for INTEGER_32'SIZE use 32;

   type LARGEST_INTEGER is range MIN_INT .. MAX_INT;

-- Address type
--
   type ADDRESS is private;

   ADDRESS_ZERO : constant ADDRESS;
   NO_ADDR      : constant ADDRESS;
   NULL_ADDRESS : constant ADDRESS;

-- Note that because ADDRESS is a private type
-- the functions "=" and "/=" are already available and
-- do not have to be explicitly defined
--
-- function "="  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
   function "<"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
   function "<=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
   function ">"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
   function ">=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;

   generic
      type TARGET is private;
   function FETCH_FROM_ADDRESS (A : ADDRESS) return TARGET;
```

```
generic
   type TARGET is private;
procedure ASSIGN_TO_ADDRESS (A : ADDRESS; T : TARGET);

-- DEC Ada floating point type declarations for the VAX
-- floating point data types

type F_FLOAT is (digits 6);
type D_FLOAT is (digits 9);
type G_FLOAT is (digits 15);

-- DEC Ada floating point type declarations for the IEEE
-- floating point data types

type IEEE_SINGLE_FLOAT is (digits 6);
type IEEE_DOUBLE_FLOAT is (digits 15);

type TYPE_CLASS is (TYPE_CLASS_ENUMERATION,
                    TYPE_CLASS_INTEGER,
                    TYPE_CLASS_FIXED_POINT,
                    TYPE_CLASS_FLOATING_POINT,
                    TYPE_CLASS_ARRAY,
                    TYPE_CLASS_RECORD,
                    TYPE_CLASS_ACCESS,
                    TYPE_CLASS_TASK,
                    TYPE_CLASS_ADDRESS);

-- AST handler type

type AST_HANDLER is limited private;

NO_AST_HANDLER : constant AST_HANDLER;

-- Non-Ada exception

NON_ADA_ERROR : exception;

-- Hardware-oriented types and functions

type BIT_ARRAY is array (INTEGER range <>) of BOOLEAN;
pragma PACK(BIT_ARRAY);

subtype BIT_ARRAY_8  is BIT_ARRAY (0 ..  7);
subtype BIT_ARRAY_16 is BIT_ARRAY (0 .. 15);
subtype BIT_ARRAY_32 is BIT_ARRAY (0 .. 31);
subtype BIT_ARRAY_64 is BIT_ARRAY (0 .. 63);

type UNSIGNED_BYTE is range 0 .. 255;
for  UNSIGNED_BYTE'SIZE use 8;

function "not" (LEFT        : UNSIGNED_BYTE) return UNSIGNED_BYTE;
function "and" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
function "or"  (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
function "xor" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;

function TO_UNSIGNED_BYTE (X : BIT_ARRAY_8) return UNSIGNED_BYTE;
function TO_BIT_ARRAY_8   (X : UNSIGNED_BYTE) return BIT_ARRAY_8;
```

```
type UNSIGNED_BYTE_ARRAY is array (INTEGER range <>) of UNSIGNED_BYTE;

type UNSIGNED_WORD is range 0 .. 65535;
for  UNSIGNED_WORD'SIZE use 16;

function "not" (LEFT        : UNSIGNED_WORD) return UNSIGNED_WORD;
function "and" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
function "or"  (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
function "xor" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;

function TO_UNSIGNED_WORD (X : BIT_ARRAY_16) return UNSIGNED_WORD;
function TO_BIT_ARRAY_16  (X : UNSIGNED_WORD) return BIT_ARRAY_16;

type UNSIGNED_WORD_ARRAY is array (INTEGER range <>) of UNSIGNED_WORD;

type UNSIGNED_LONGWORD is range -2_147_483_648 .. 2_147_483_647;
for  UNSIGNED_LONGWORD'SIZE use 32;

function "not" (LEFT        : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "and" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "or"  (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "xor" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;

function TO_UNSIGNED_LONGWORD (X : BIT_ARRAY_32) return UNSIGNED_LONGWORD;
function TO_BIT_ARRAY_32 (X : UNSIGNED_LONGWORD) return BIT_ARRAY_32;

type UNSIGNED_LONGWORD_ARRAY is
   array (INTEGER range <>) of UNSIGNED_LONGWORD;

type UNSIGNED_QUADWORD is
   record
      L0 : UNSIGNED_LONGWORD;
      L1 : UNSIGNED_LONGWORD;
   end record;
for  UNSIGNED_QUADWORD'SIZE use 64;
for UNSIGNED_QUADWORD use
   record
      at mod 8;
   end record;

function "not" (LEFT        : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
function "and" (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
function "or"  (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
function "xor" (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;

function TO_UNSIGNED_QUADWORD (X : BIT_ARRAY_64) return UNSIGNED_QUADWORD;
function TO_BIT_ARRAY_64 (X : UNSIGNED_QUADWORD) return BIT_ARRAY_64;

type UNSIGNED_QUADWORD_ARRAY is
   array (INTEGER range <>) of UNSIGNED_QUADWORD;

function TO_ADDRESS  (X : INTEGER)              return ADDRESS;
function TO_ADDRESS  (X : UNSIGNED_LONGWORD)    return ADDRESS;
function TO_ADDRESS  (X : {universal_integer})  return ADDRESS;

function TO_INTEGER           (X : ADDRESS)     return INTEGER;
function TO_UNSIGNED_LONGWORD (X : ADDRESS)     return UNSIGNED_LONGWORD;
```

```
   function TO_UNSIGNED_LONGWORD (X : AST_HANDLER) return UNSIGNED_LONGWORD;

-- Conventional names for static subtypes of type UNSIGNED_LONGWORD

   subtype UNSIGNED_1  is UNSIGNED_LONGWORD range 0 .. 2** 1-1;
   subtype UNSIGNED_2  is UNSIGNED_LONGWORD range 0 .. 2** 2-1;
   subtype UNSIGNED_3  is UNSIGNED_LONGWORD range 0 .. 2** 3-1;
   subtype UNSIGNED_4  is UNSIGNED_LONGWORD range 0 .. 2** 4-1;
   subtype UNSIGNED_5  is UNSIGNED_LONGWORD range 0 .. 2** 5-1;
   subtype UNSIGNED_6  is UNSIGNED_LONGWORD range 0 .. 2** 6-1;
   subtype UNSIGNED_7  is UNSIGNED_LONGWORD range 0 .. 2** 7-1;
   subtype UNSIGNED_8  is UNSIGNED_LONGWORD range 0 .. 2** 8-1;
   subtype UNSIGNED_9  is UNSIGNED_LONGWORD range 0 .. 2** 9-1;
   subtype UNSIGNED_10 is UNSIGNED_LONGWORD range 0 .. 2**10-1;
   subtype UNSIGNED_11 is UNSIGNED_LONGWORD range 0 .. 2**11-1;
   subtype UNSIGNED_12 is UNSIGNED_LONGWORD range 0 .. 2**12-1;
   subtype UNSIGNED_13 is UNSIGNED_LONGWORD range 0 .. 2**13-1;
   subtype UNSIGNED_14 is UNSIGNED_LONGWORD range 0 .. 2**14-1;
   subtype UNSIGNED_15 is UNSIGNED_LONGWORD range 0 .. 2**15-1;
   subtype UNSIGNED_16 is UNSIGNED_LONGWORD range 0 .. 2**16-1;
   subtype UNSIGNED_17 is UNSIGNED_LONGWORD range 0 .. 2**17-1;
   subtype UNSIGNED_18 is UNSIGNED_LONGWORD range 0 .. 2**18-1;
   subtype UNSIGNED_19 is UNSIGNED_LONGWORD range 0 .. 2**19-1;
   subtype UNSIGNED_20 is UNSIGNED_LONGWORD range 0 .. 2**20-1;
   subtype UNSIGNED_21 is UNSIGNED_LONGWORD range 0 .. 2**21-1;
   subtype UNSIGNED_22 is UNSIGNED_LONGWORD range 0 .. 2**22-1;
   subtype UNSIGNED_23 is UNSIGNED_LONGWORD range 0 .. 2**23-1;
   subtype UNSIGNED_24 is UNSIGNED_LONGWORD range 0 .. 2**24-1;
   subtype UNSIGNED_25 is UNSIGNED_LONGWORD range 0 .. 2**25-1;
   subtype UNSIGNED_26 is UNSIGNED_LONGWORD range 0 .. 2**26-1;
   subtype UNSIGNED_27 is UNSIGNED_LONGWORD range 0 .. 2**27-1;
   subtype UNSIGNED_28 is UNSIGNED_LONGWORD range 0 .. 2**28-1;
   subtype UNSIGNED_29 is UNSIGNED_LONGWORD range 0 .. 2**29-1;
   subtype UNSIGNED_30 is UNSIGNED_LONGWORD range 0 .. 2**30-1;
   subtype UNSIGNED_31 is UNSIGNED_LONGWORD range 0 .. 2**31-1;

-- Function for obtaining global symbol values

   function IMPORT_VALUE (SYMBOL : STRING) return UNSIGNED_LONGWORD;

private

   -- Not shown

end SYSTEM;
```

F.3.3 The Package System on ULTRIX Systems

```
package SYSTEM is

    type NAME is (RISC_ULTRIX);
    for NAME use (RISC_ULTRIX => 6);

    SYSTEM_NAME   : constant NAME := RISC_ULTRIX;
    STORAGE_UNIT  : constant := 8;
    MEMORY_SIZE   : constant := 2**31-1;
    MAX_INT       : constant := 2**31-1;
    MIN_INT       : constant := -(2**31);
    MAX_DIGITS    : constant := 15;
    MAX_MANTISSA  : constant := 31;
    FINE_DELTA    : constant := 2.0**(-31);
    TICK          : constant := 3.906 * 10.0**(-3);

    subtype PRIORITY is INTEGER range 0 .. 15;

--  Address type
--
    type ADDRESS is private;

    ADDRESS_ZERO : constant ADDRESS;

    function "+" (LEFT : ADDRESS; RIGHT : INTEGER) return ADDRESS;
    function "+" (LEFT : INTEGER; RIGHT : ADDRESS) return ADDRESS;
    function "-" (LEFT : ADDRESS; RIGHT : ADDRESS) return INTEGER;
    function "-" (LEFT : ADDRESS; RIGHT : INTEGER) return ADDRESS;

--  function "="  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
--  function "/=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function "<"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function "<=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function ">"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function ">=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;

--  Note that because ADDRESS is a private type
--  the functions "=" and "/=" are already available and
--  do not have to be explicitly defined

    generic
        type TARGET is private;
    function FETCH_FROM_ADDRESS (A : ADDRESS) return TARGET;

    generic
        type TARGET is private;
    procedure ASSIGN_TO_ADDRESS (A : ADDRESS; T : TARGET);

--  DEC Ada floating point type declarations for the IEEE
--  floating point data types

    type IEEE_SINGLE_FLOAT is (digits 6);
    type IEEE_DOUBLE_FLOAT is (digits 15);
```

```ada
    type TYPE_CLASS is (TYPE_CLASS_ENUMERATION,
                        TYPE_CLASS_INTEGER,
                        TYPE_CLASS_FIXED_POINT,
                        TYPE_CLASS_FLOATING_POINT,
                        TYPE_CLASS_ARRAY,
                        TYPE_CLASS_RECORD,
                        TYPE_CLASS_ACCESS,
                        TYPE_CLASS_TASK,
                        TYPE_CLASS_ADDRESS);

--  Non-Ada exception

    NON_ADA_ERROR : exception;

--  Hardware-oriented types and functions

    type    BIT_ARRAY is array (INTEGER range <>) of BOOLEAN;
    pragma  PACK(BIT_ARRAY);

    subtype BIT_ARRAY_8  is BIT_ARRAY (0 ..  7);
    subtype BIT_ARRAY_16 is BIT_ARRAY (0 .. 15);
    subtype BIT_ARRAY_32 is BIT_ARRAY (0 .. 31);
    subtype BIT_ARRAY_64 is BIT_ARRAY (0 .. 63);

    type UNSIGNED_BYTE   is range 0 .. 255;
    for  UNSIGNED_BYTE'SIZE  use 8;

    function "not" (LEFT        : UNSIGNED_BYTE) return UNSIGNED_BYTE;
    function "and" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
    function "or"  (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
    function "xor" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;

    function TO_UNSIGNED_BYTE (X : BIT_ARRAY_8) return UNSIGNED_BYTE;
    function TO_BIT_ARRAY_8 (X : UNSIGNED_BYTE) return BIT_ARRAY_8;

    type UNSIGNED_BYTE_ARRAY is array (INTEGER range <>) of UNSIGNED_BYTE;

    type UNSIGNED_WORD   is range 0 .. 65535;
    for  UNSIGNED_WORD'SIZE  use 16;

    function "not" (LEFT        : UNSIGNED_WORD) return UNSIGNED_WORD;
    function "and" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
    function "or"  (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
    function "xor" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;

    function TO_UNSIGNED_WORD (X : BIT_ARRAY_16) return UNSIGNED_WORD;
    function TO_BIT_ARRAY_16 (X : UNSIGNED_WORD) return BIT_ARRAY_16;

    type UNSIGNED_WORD_ARRAY is array (INTEGER range <>) of UNSIGNED_WORD;

    type UNSIGNED_LONGWORD is range MIN_INT .. MAX_INT;
    for  UNSIGNED_LONGWORD'SIZE use 32;

    function "not" (LEFT        : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
    function "and" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
    function "or"  (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
    function "xor" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
```

```
    function TO_UNSIGNED_LONGWORD (X : BIT_ARRAY_32)
       return UNSIGNED_LONGWORD;
    function TO_BIT_ARRAY_32 (X : UNSIGNED_LONGWORD) return BIT_ARRAY_32;

    type UNSIGNED_LONGWORD_ARRAY is
       array (INTEGER range <>) of UNSIGNED_LONGWORD;

    type UNSIGNED_QUADWORD is record
        L0 : UNSIGNED_LONGWORD;
        L1 : UNSIGNED_LONGWORD;
        end record;
    for  UNSIGNED_QUADWORD'SIZE use 64;

    function "not" (LEFT        : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
    function "and" (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
    function "or"  (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
    function "xor" (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;

    function TO_UNSIGNED_QUADWORD (X : BIT_ARRAY_64)
       return UNSIGNED_QUADWORD;
    function TO_BIT_ARRAY_64 (X : UNSIGNED_QUADWORD) return BIT_ARRAY_64;

    type UNSIGNED_QUADWORD_ARRAY is
       array (INTEGER range <>) of UNSIGNED_QUADWORD;

    function TO_ADDRESS  (X : INTEGER)                return ADDRESS;
    function TO_ADDRESS  (X : UNSIGNED_LONGWORD)      return ADDRESS;
    function TO_ADDRESS  (X : {universal_integer})    return ADDRESS;

    function TO_INTEGER           (X : ADDRESS)       return INTEGER;
    function TO_UNSIGNED_LONGWORD (X : ADDRESS)       return UNSIGNED LONGWORD;

--  Conventional names for static subtypes of type UNSIGNED_LONGWORD

    subtype UNSIGNED_1  is UNSIGNED_LONGWORD range 0 .. 2** 1-1;
    subtype UNSIGNED_2  is UNSIGNED_LONGWORD range 0 .. 2** 2-1;
    subtype UNSIGNED_3  is UNSIGNED_LONGWORD range 0 .. 2** 3-1;
    subtype UNSIGNED_4  is UNSIGNED_LONGWORD range 0 .. 2** 4-1;
    subtype UNSIGNED_5  is UNSIGNED_LONGWORD range 0 .. 2** 5-1;
    subtype UNSIGNED_6  is UNSIGNED_LONGWORD range 0 .. 2** 6-1;
    subtype UNSIGNED_7  is UNSIGNED_LONGWORD range 0 .. 2** 7-1;
    subtype UNSIGNED_8  is UNSIGNED_LONGWORD range 0 .. 2** 8-1;
    subtype UNSIGNED_9  is UNSIGNED_LONGWORD range 0 .. 2** 9-1;
    subtype UNSIGNED_10 is UNSIGNED_LONGWORD range 0 .. 2**10-1;
    subtype UNSIGNED_11 is UNSIGNED_LONGWORD range 0 .. 2**11-1;
    subtype UNSIGNED_12 is UNSIGNED_LONGWORD range 0 .. 2**12-1;
    subtype UNSIGNED_13 is UNSIGNED_LONGWORD range 0 .. 2**13-1;
    subtype UNSIGNED_14 is UNSIGNED_LONGWORD range 0 .. 2**14-1;
    subtype UNSIGNED_15 is UNSIGNED_LONGWORD range 0 .. 2**15-1;
    subtype UNSIGNED_16 is UNSIGNED_LONGWORD range 0 .. 2**16-1;
    subtype UNSIGNED_17 is UNSIGNED_LONGWORD range 0 .. 2**17-1;
    subtype UNSIGNED_18 is UNSIGNED_LONGWORD range 0 .. 2**18-1;
    subtype UNSIGNED_19 is UNSIGNED_LONGWORD range 0 .. 2**19-1;
    subtype UNSIGNED_20 is UNSIGNED_LONGWORD range 0 .. 2**20-1;
```

```
    subtype UNSIGNED_21 is UNSIGNED_LONGWORD range 0 .. 2**21-1;
    subtype UNSIGNED_22 is UNSIGNED_LONGWORD range 0 .. 2**22-1;
    subtype UNSIGNED_23 is UNSIGNED_LONGWORD range 0 .. 2**23-1;
    subtype UNSIGNED_24 is UNSIGNED_LONGWORD range 0 .. 2**24-1;
    subtype UNSIGNED_25 is UNSIGNED_LONGWORD range 0 .. 2**25-1;
    subtype UNSIGNED_26 is UNSIGNED_LONGWORD range 0 .. 2**26-1;
    subtype UNSIGNED_27 is UNSIGNED_LONGWORD range 0 .. 2**27-1;
    subtype UNSIGNED_28 is UNSIGNED_LONGWORD range 0 .. 2**28-1;
    subtype UNSIGNED_29 is UNSIGNED_LONGWORD range 0 .. 2**29-1;
    subtype UNSIGNED_30 is UNSIGNED_LONGWORD range 0 .. 2**30-1;
    subtype UNSIGNED_31 is UNSIGNED_LONGWORD range 0 .. 2**31-1;

--  Function for obtaining global symbol values

    function IMPORT_VALUE (SYMBOL : STRING) return UNSIGNED_LONGWORD;

private

    -- Not shown

end SYSTEM;
```

F.4 Restrictions on Representation Clauses

The representation clauses allowed in DEC Ada are length, enumeration, record representation, and address clauses.

In DEC Ada, a representation clause for a generic formal type or a type that depends on a generic formal type is not allowed. In addition, a representation clause for a composite type that has a component or subcomponent of a generic formal type or a type derived from a generic formal type is not allowed.

F.5 Restrictions on Unchecked Type Conversions

DEC Ada supports the generic function UNCHECKED_CONVERSION with the following restrictions on the class of types involved:

- The actual subtype corresponding to the formal type TARGET must not be an unconstrained array type.
- The actual subtype corresponding to the formal type TARGET must not be an unconstrained type with discriminants.

Further, when the target type is a type with discriminants, the value resulting from a call of the conversion function resulting from an instantiation of UNCHECKED_CONVERSION is checked to ensure that the discriminants satisfy the constraints of the actual subtype.

If the size of the source value is greater than the size of the target subtype, then the high order bits of the value are ignored (truncated); if the size of the source value is less than the size of the target subtype, then the value is extended with zero bits to form the result value.

F.6 Conventions for Implementation-Generated Names Denoting Implementation-Dependent Components in Record Representation Clauses

DEC Ada does not allocate implementation-dependent components in records.

F.7 Interpretation of Expressions Appearing in Address Clauses

Expressions appearing in address clauses must be of the type ADDRESS defined in the package SYSTEM (see 13.7a.1 and F.3). In DEC Ada, values of type SYSTEM.ADDRESS are interpreted as virtual addresses in the machine's address space.

DEC Ada allows address clauses for objects and imported subprograms (see 13.5).

DEC Ada does not support interrupts as defined in section 13.5.1.

On OpenVMS systems, DEC Ada provides the pragma AST_ENTRY and the AST_ENTRY attribute as alternative mechanisms for handling asynchronous interrupts from the OpenVMS operating system (see 9.12a).

For information on handling ULTRIX signals, see the *DEC Ada Run-Time Reference Manual for ULTRIX Systems.*

F.8 Implementation-Dependent Characteristics of Input-Output Packages

In addition to the standard predefined input-output packages (SEQUENTIAL_IO, DIRECT_IO, TEXT_IO, and IO_EXCEPTIONS), DEC Ada provides packages for handling sequential and direct files with mixed-type elements:

- SEQUENTIAL_MIXED_IO (see 14.2b.4).
- DIRECT_MIXED_IO (see 14.2b.6).

DEC Ada does not provide the low level input-output package described in this section.

As specified in section 14.4, DEC Ada raises the following language-defined exceptions for error conditions that occur during input-output operations: STATUS_ERROR, MODE_ERROR, NAME_ERROR, USE_ERROR, END_ERROR, DATA_ERROR, and LAYOUT_ERROR. DEC Ada does not raise the language-defined exception DEVICE_ERROR; device-related errors cause the exception USE_ERROR to be raised.

The exception USE_ERROR is raised under the following conditions:

- If the capacity of the external file has been exceeded.
- In all CREATE operations if the mode specified is IN_FILE.
- In all CREATE operations if the file attributes specified by the FORM parameter are not supported by the package.
- In all CREATE, OPEN, DELETE, and RESET operations if, for the specified mode, the environment does not support the operation for an external file.
- In all NAME operations if the file has no name.
- In the SET_LINE_LENGTH and SET_PAGE_LENGTH operations on text files if the lengths specified are inappropriate for the external file.
- In text files if an operation is attempted that is not possible for reasons that depend on characteristics of the external file.

DEC Ada provides other input-output packages that are available on specific systems. The following sections outline those packages. The following sections also give system-specific information about the overall set of DEC Ada input-output packages and input-output exceptions.

F.8.1 DEC Ada Input-Output Packages on OpenVMS Systems

On OpenVMS systems, the DEC Ada predefined packages and their operations are implemented using OpenVMS Record Management Services (RMS) file organizations and facilities. To give users the maximum benefit of the underlying RMS input-output facilities, DEC Ada provides the following *OpenVMS-specific packages:*

- RELATIVE_IO (see 14.2a.3).
- INDEXED_IO (see 14.2a.5).
- RELATIVE_MIXED_IO (see 14.2b.8).
- INDEXED_MIXED_IO (see 14.2b.10).
- AUX_IO_EXCEPTIONS (see 14.5a).

The following sections summarize the implementation-dependent characteristics of the DEC Ada input-output packages. The *DEC Ada Run-Time Reference Manual for OpenVMS Systems* discusses these characteristics in more detail.

F.8.1.1 Interpretation of the FORM Parameter on OpenVMS Systems

On OpenVMS systems, the value of the FORM parameter may be a string of statements of the OpenVMS Record Management Services (RMS) File Definition Language (FDL), or it may be a string referring to a text file of FDL statements (called an FDL file).

FDL is a special-purpose OpenVMS language for writing file specifications. These specifications are then used by DEC Ada run-time routines to create or open files. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for the rules governing the FORM parameter and for a general description of FDL. See the *Guide to OpenVMS File Applications* and the *OpenVMS Record Management Utilities Reference Manual* for complete information on FDL.

On OpenVMS systems, each input-output package has a default string of FDL statements that is used to open or create a file. Thus, in general, specification of a FORM parameter is not necessary: it is never necessary in an OPEN procedure; it may be necessary in a CREATE procedure. The packages for which a value for the FORM parameter must be specified in a CREATE procedure are as follows:

- The packages DIRECT_IO and RELATIVE_IO require that a maximum element (record) size be specified in the FORM parameter if the item with which the package is instantiated is unconstrained.
- The packages DIRECT_MIXED_IO and RELATIVE_MIXED_IO require that a maximum element (record) size be specified in the FORM parameter.
- The packages INDEXED_IO and INDEXED_MIXED_IO require that information about keys be specified in the FORM parameter.

Any explicit FORM specification supersedes the default attributes of the governing input-output package. The *DEC Ada Run-Time Reference Manual for OpenVMS Systems* describes the default external file attributes of each input-output package.

The use of the FORM parameter is described for each input-output package in chapter 14. For information on the default FORM parameters for each DEC Ada input-output package and for information on using the FORM parameter to specify external file attributes, see the *DEC Ada Run-Time Reference Manual for OpenVMS Systems*. For information on FDL, see the *Guide to OpenVMS File Applications* and the *OpenVMS Record Management Utilities Reference Manual*.

F.8.1.2 Input-Output Exceptions on OpenVMS Systems

In addition to the DEC Ada exceptions that apply on all systems, the following also apply on OpenVMS systems:

- The DEC Ada exceptions LOCK_ERROR, EXISTENCE_ERROR, and KEY_ERROR are raised for relative and indexed input-output operations.
- The exception USE_ERROR is raised as follows in relative and indexed files:
  - In the WRITE operations on relative or indexed files if the element in the position indicated has already been written.
  - In the DELETE_ELEMENT operations on relative and indexed files if the current element is undefined at the start of the operation.
  - In the UPDATE operations on indexed files if the current element is undefined or if the specified key violates the external file attributes.
- The exception NAME_ERROR is raised as specified in section 14.4: by a call of a CREATE or OPEN procedure if the string given fo: the NAME parameter does not allow the identification of an external file. On OpenVMS systems, the value of a NAME parameter can be a string that denotes a OpenVMS file specification or a OpenVMS logical name (in either case, the string names an external file). For a CREATE procedure, the value of a NAME parameter can also be a null string, in which case it names a temporary external file that is deleted when the main program exits. The *DEC Ada Run-Time Reference Manual for OpenVMS Systems* explains the naming of external files in more detail.
- The exception LAYOUT_ERROR is raised as specified in section 14.4: in text input-output by COL, LINE, or PAGE if the value returned exceeds COUNT'LAST. The exception LAYOUT_ERROR is also raised on output by an attempt to set column or line numbers in excess of specified maximum line or page lengths, and by attempts to PUT too many characters to a string. In the DEC Ada mixed input-output packages, the exception LAYOUT_ERROR is raised by GET_ITEM if no more items can be read from the file buffer; it is raised by PUT_ITEM if the current position exceeds the file buffer size.

F.8.2 Input-Output Packages on ULTRIX Systems

On ULTRIX systems, the DEC Ada predefined packages and their operations are implemented using ULTRIX file facilities. DEC Ada provides no additional input-output packages specifically related to ULTRIX systems.

The following sections summarize the ULTRIX-specific characteristics of the DEC Ada input-output packages. The *DEC Ada Run-Time Reference Manual for ULTRIX Systems* discusses these characteristics in more detail.

F.8.2.1 Interpretation of the FORM Parameter on ULTRIX Systems

On ULTRIX systems, the value of the FORM parameter must be a character string, defined as follows:

```
string       ::== "[field {,field}]"

field        ::== field_id => field_value

field_id     ::== BUFFER_SIZE | ELEMENT_SIZE | FILE_DESCRIPTOR

field_value ::== digit {digit}
```

Depending on the fields specified, the value of the FORM parameter may represent one or more of the following:

- The size of the buffer used during file operations. The field value specifies the number of bytes in the buffer.
- The maximum element size for a direct file. The field value specifies the maximum number of bytes in the element.
- An ULTRIX file descriptor for the Ada file being opened. The ULTRIX file descriptor must be open.

  If the file descriptor is not open, or if it refers to an Ada file that is already open, then the exception USE_ERROR is raised. Note that the file descriptor option can be used only in the FORM parameter of an OPEN procedure.

Each input-output package has an implementation-defined value form string that is used to open or create a file. Thus, in general, specification of a FORM parameter is not necessary. The packages for which a value for the FORM parameter must be specified in a CREATE procedure are as follows:

- The package DIRECT_IO requires that a maximum element size be specified in the FORM parameter if the item with which the package is instantiated is unconstrained.
- The package DIRECT_MIXED_IO requires that a maximum element size be specified in the FORM parameter.

The use of the FORM parameter is described for each input-output package in chapter 14. For information on using the FORM parameter to specify external file attributes, see the *DEC Ada Run-Time Reference Manual for ULTRIX Systems*.

F.8.2.2 Input-Output Exceptions on ULTRIX Systems

In addition to the DEC Ada exceptions that apply on all systems, the following also apply on ULTRIX systems:

- The exception NAME_ERROR is raised as specified in section 14.4: by a call of a CREATE or OPEN procedure if the string given for the NAME parameter does not allow the identification of an external file. On ULTRIX systems, the value of a NAME parameter can be a string that denotes an ULTRIX file specification. For a CREATE procedure, the value of a NAME parameter can also be a null string, in which case it names a temporary external file that is deleted when the main program exits. The *DEC Ada Run-Time Reference Manual for ULTRIX Systems* explains the naming of external files in more detail.
- The exception LAYOUT_ERROR is raised as specified in section 14.4: in text input-output by COL, LINE, or PAGE if the value returned exceeds COUNT'LAST. The exception LAYOUT_ERROR is also raised on output by an attempt to set column or line numbers in excess of specified maximum line or page lengths, and by attempts to PUT too many characters to a string. In the DEC Ada mixed input-output packages, the exception LAYOUT_ERROR is raised by GET_ITEM if no more items can be read from the file buffer; it is raised by PUT_ITEM if the current position exceeds the file buffer size.

F.9 Other Implementation Characteristics

Implementation characteristics relating to the definition of a main program, various numeric ranges, and implementation limits are summarized in the following sections.

F.9.1 Definition of a Main Program

DEC Ada permits a library unit to be used as a main program under the following conditions:

- If it is a procedure with no formal parameters.

  On OpenVMS systems, the status returned to the OpenVMS environment upon normal completion of the procedure is the value 1.

  On ULTRIX systems, the status returned to the ULTRIX environment upon normal completion of the procedure is the value 0.

- If it is a function with no formal parameters whose returned value is of a discrete type. In this case, the status returned to the operating-system environment upon normal completion of the function is the function value.
- If it is a procedure declared with the pragma EXPORT_VALUED_PROCEDURE, and it has one formal **out** parameter that is of a discrete type. In this case, the status returned to the operating-system environment upon normal completion of the procedure is the value of the first (and only) parameter.

Note that when a main function or a main procedure declared with the pragma EXPORT_VALUED_PROCEDURE returns a discrete value whose size is less than 32 bits, the value is zero- or sign-extended as appropriate.

F.9.2 Values of Integer Attributes

The ranges of values for integer types declared in the package STANDARD are as follows:

Integer type	Range	Systems on which it applies
SHORT_SHORT_INTEGER	–128 .. 127	All
SHORT_INTEGER	–32768 .. 32767	All
INTEGER	–2147483648 .. 2147483647	All
LONG_INTEGER	–2147483648 .. 2147483647	OpenVMS

For the applicable input-output packages, the ranges of values for the types COUNT and POSITIVE_COUNT are as follows:

COUNT	0 .. INTEGER'LAST
POSITIVE_COUNT	1 .. INTEGER'LAST

For the package TEXT_IO, the range of values for the type FIELD is as follows:

FIELD	0 .. INTEGER'LAST

F.9.3 Values of Floating Point Attributes

DEC Ada provides a number of predefined floating point types, as shown in the following table:

Type	Representation	Systems on which it applies	Section
FLOAT	F_floating IEEE single float	All OpenVMS[1] ULTRIX, OpenVMS AXP[2]	3.5.7
LONG_FLOAT	D_floating or G_floating IEEE double float	All OpenVMS[1] ULTRIX, OpenVMS AXP[2]	3.5.7
LONG_LONG_FLOAT	H_floating	OpenVMS VAX	3.5.7
F_FLOAT	F_floating	All OpenVMS	3.5.7
D_FLOAT	D_floating	All OpenVMS	3.5.7
G_FLOAT	G_floating	All OpenVMS	3.5.7
H_FLOAT	H_floating	OpenVMS VAX	3.5.7
IEEE_SINGLE_FLOAT	IEEE single float	ULTRIX, OpenVMS AXP[2]	3.5.7
IEEE_DOUBLE_FLOAT	IEEE double float	ULTRIX, OpenVMS AXP[2]	3.5.7

[1]When the value of the pragma FLOAT_REPRESENTATION is VAX_FLOAT.

[2]When the value of the pragma FLOAT_REPRESENTATION is IEEE_FLOAT.

The values of the floating point attributes for the different floating point representations appear in the following tables.

F.9.3.1 F_floating Characteristics

Attribute	F_floating value and approximate decimal equivalent (where applicable)
DIGITS	6
MANTISSA	21
EMAX	84
EPSILON	16#0.1000_000#e-4
approximately	9.53674E-07
SMALL	16#0.8000_000#e-21
approximately	2.58494E-26
LARGE	16#0.FFFF_F80#e+21
approximately	1.93428E+25
SAFE_EMAX	127
SAFE_SMALL	16#0.1000_000#e-31
approximately	2.93874E-39
SAFE_LARGE	16#0.7FFF_FC0#e+32
approximately	1.70141E+38
FIRST	-16#0.7FFF_FF8#e+32
approximately	-1.70141E+38
LAST	16#0.7FFF_FF8#e+32
approximately	1.70141E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	24
MACHINE_EMAX	127
MACHINE_EMIN	-127
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

F.9.3.2 D_floating Characteristics

Attribute	D_floating value and approximate decimal equivalent (where applicable)
DIGITS	9
MANTISSA	31
EMAX	124
EPSILON	16#0.4000_0000_0000_000#e–7
approximately	9.3132257461548E–10
SMALL	16#0.8000_0000_0000_000#e–31
approximately	2.3509887016446E–38
LARGE	16#0.FFFF_FFFE_0000_000#e+31
approximately	2.1267647922655E+37
SAFE_EMAX	127
SAFE_SMALL	16#0.1000_0000_0000_000#e–31
approximately	2.9387358770557E–39
SAFE_LARGE	16#0.7FFF_FFFF_0000_000#e+32
approximately	1.7014118338124E+38
FIRST	–16#0.7FFF_FFFF_FFFF_FF8#e+32
approximately	–1.7014118346047E+38
LAST	16#0.7FFF_FFFF_FFFF_FF8#e+32
approximately	1.7014118346047E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	56
MACHINE_EMAX	127
MACHINE_EMIN	–127
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

F.9.3.3 G_floating Characteristics

Attribute	G_floating value and approximate decimal equivalent (where applicable)
DIGITS	15
MANTISSA	51
EMAX	204
EPSILON	16#0.4000_0000_0000_00#e–12
approximately	8.881784197001E–16
SMALL	16#0.8000_0000_0000_00#e–51
approximately	1.944692274332E–62
LARGE	16#0.FFFF_FFFF_FFFF_E0#e+51
approximately	2.571100870814E+61
SAFE_EMAX	1023
SAFE_SMALL	16#0.1000_0000_0000_00#e–255
approximately	5.562684646268E–309
SAFE_LARGE	16#0.7FFF_FFFF_FFFF_F0#e+256
approximately	8.988465674312E+307
FIRST	–16#0.7FFF_FFFF_FFFF_FC#e+256
approximately	–8.988465674312E+307
LAST	16#0.7FFF_FFFF_FFFF_FC#e+256
approximately	8.988465674312E+307
MACHINE_RADIX	2
MACHINE_MANTISSA	53
MACHINE_EMAX	1023
MACHINE_EMIN	–1023
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

F.9.3.4 H_floating Characteristics

Attribute	H_floating value and approximate decimal equivalent (where applicable)
DIGITS	33
MANTISSA	111
EMAX	444
EPSILON	16#0.4000_0000_0000_0000_0000_0000_0000_0#e–27
approximately	7.7037197775489434122239117703397E–34
SMALL	16#0.8000_0000_0000_0000_0000_0000_0000_0#e–111
approximately	1.1006568214637918210934318020936E–134
LARGE	16#0.FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFE_0#e+111
approximately	4.5427420268475430659332737993000E+133
SAFE_EMAX	16383
SAFE_SMALL	16#0.1000_0000_0000_0000_0000_0000_0000_0#e–4095
approximately	8.4052578577802337656566945433044E–4933
SAFE_LARGE	16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_0#e+4096
approximately	5.9486574767861588254287966331400E+4931
FIRST	–16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_C#e+4096
approximately	–5.9486574767861588254287966331400E+4931
LAST	16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_C#e+4096
approximately	5.9486574767861588254287966331400E+4931
MACHINE_RADIX	2
MACHINE_MANTISSA	113
MACHINE_EMAX	16383
MACHINE_EMIN	–16383
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

F.9.3.5 IEEE Single Float Characteristics

Attribute	IEEE single float value and approximate decimal equivalent (where applicable)
DIGITS	6
MANTISSA	21
EMAX	84
EPSILON	16#0.1000_000#e–4
approximately	9.53674E-07
SMALL	16#0.8000_000#e–21
approximately	2.5849E–26
LARGE	16#0.FFFF_F80#E+21
approximately	1.93428E+25
SAFE_EMAX	125
SAFE_SMALL	
approximately	1.17549E–38
SAFE_LARGE	
approximately	4.25353E+37
FIRST	
approximately	–3.40282E+38
LAST	
approximately	3.40282E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	24
MACHINE_EMAX	128
MACHINE_EMIN	–125
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

F.9.3.6 IEEE Double Float Characteristics

Attribute	IEEE double float value and approximate decimal equivalent (where applicable)
DIGITS	15
MANTISSA	51
EMAX	204
EPSILON	
approximately	8.8817841970012E-16
SMALL	
approximately	1.9446922743316E-62
LARGE	
approximately	2.5711008708144E+61
SAFE_EMAX	1021
SAFE_SMALL	
approximately	2.22507385850720E-308
SAFE_LARGE	
approximately	2.2471164185779E+307
FIRST	
approximately	-1.7976931348623E+308
LAST	
approximately	1.7976931348623E+308
MACHINE_RADIX	2
MACHINE_MANTISSA	53
MACHINE_EMAX	1024
MACHINE_EMIN	-1021
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

F.9.4 Attributes of Type DURATION

The values of the significant attributes of the type DURATION are as follows:

DURATION'DELTA	0.0001
DURATION'SMALL	2^{-14}
DURATION'FIRST	-131072.0000

DURATION'LAST	131071.9999
DURATION'LARGE	131071.9999

F.9.5 Implementation Limits

Limit	DEC systems on which it applies	Value
Maximum number of formal parameters in a subprogram or entry declaration that are of an unconstrained record type	All	32
Maximum identifier length (number of characters)	All	255
Maximum number of characters in a source line	All	255
Maximum number of discriminants for a record type	All	245
Maximum number of formal parameters in an entry or subprogram declaration	All	246
Maximum number of dimensions in an array type	All	255
Maximum number of library units and subunits in a compilation closure[1]	All	4095
Maximum number of library units and subunits in an execution closure[2]	All	16383
Maximum number of objects declared with the pragma COMMON_OBJECT or PSECT_OBJECT	All	32757
Maximum number of enumeration literals in an enumeration type definition	All	65535
Maximum number of lines in a source file	All	65534
Maximum number of bits in any object	All	$2^{31} - 1$
Maximum size of the static portion of a stack frame (approximate)	All	2^{30}

[1]The compilation closure of a given unit is the total set of units that the given unit depends on, directly and indirectly.

[2]The execution closure of a given unit is the compilation closure plus all associated secondary units (library bodies and subunits).